Aproach to #SAT in O(n)

Juan Manuel Dato Ruiz

Abstract
How do you calculate the i-th number of Fibonacci in a Turing Machine? There are two easy philosophies when your entry are the digits of a number in a specific base. The easiest one is described by a loop and two temporal variables which store two numbers in each step. This way requires an exponential number of steps, because the number of steps is proportional with the number put in the entry.

But there is other philosophy which gives us a result faster: if we corroborate the ratio between the biggest consecutive numbers of Fibonacci is the Golden Ratio, then the Fibonacci expression can be written in a way

$$F_n \sim K_1 \times \Phi^n$$

Even more, considering the sucession is strictly increasing then the error will be expressed in the same way with other real number lower in absolute value.

If we decide to use this other philosophy, the numbers of steps will be constant, but the result will depend of the number of digits we required for the parameters of the algorithm (i.e. $\mathbf{K_1}$ and $\mathbf{\Phi}$). So if the precision of the real numbers was not too good, then the result in great values could be inexact.

That was the real plot of this document: a good example which showed us that if our solutions were based in something similar to Fibonacci numbers, then we would get two results. One fast and appropiated in Formal Computing and one well defined and exact in Constructive Computing.

Introducción.
Existen diversos mecanismos para determinar cuáles son las soluciones de una fórmula del álgebra de Bool. Pero la operación que deja obsoleta a todas las demás es la que nos ofrece el número de soluciones que satisfacen la fórmula. Es decir, cuantas secuencias independientes de asignaciones sobre las variables booleanas consiguen satisfacer la igualdad propuesta. Este documento, paradógicamente, en combinación con otros documentos presentados, podría incluso invitar a contabilizar el número de soluciones de una fórmula dentro del $\mathbf{Z_n}$, siempre con la intención de alcanzar la O(**n**) con **n** el tamaño de la entrada, o por lo menos asegurar la cota polinomial.

Desarrollo.
Nosotros partiremos del lugar en donde se deja el estudio del álgebra de Bool para convertirlo al formato producto de alternancias. Más concretamente, nos planteamos para los conjuntos $\mathbf{A_k}$ y **B** definidos por (1), cuántos posibles conjuntos **B** se pueden definir a partir de los conjuntos $\mathbf{A_k}$.

$$\begin{gathered} \forall k \in \mathbb{N}\, A_k \in 2^{\mathbb{N}} \exists B \in 2^{\mathbb{N}}\, donde: \\ 1. \forall x \exists k\, x \in B \rightarrow x \in A_k \\ 2. \forall x \forall y \forall k\, x \in A_k \wedge y \in A_k \wedge x \in B \wedge y \in B \rightarrow x = y \end{gathered} \quad (1)$$

Bajo ese formato es fácil encontrar equivalencias con fórmulas de la lógica, concretamente, en (2) vemos una manera de asociar los A_k definidos en (1) con el sumador completo.

$$\begin{gathered} A_1 = \{1, 2, 3\} \\ A_2 = \{4, 2, 5\} \\ A_3 = \{3, 5, 6\} \\ p \sim 1 \notin B \\ q \sim 4 \notin B \\ p \wedge q \sim 2 \in B \\ p \oplus q \sim 6 \notin B \end{gathered} \quad (2)$$

Poder crear equivalencias de $\mathbf{Z_n}$ para cualquier **n** permitiría generalizar las operaciones vertidas en

este documento. Sin embargo, nuestro objeto no es modelar mediante el formato (1), sino reconvertir ese formato a otro más adecuado para calcular el número de soluciones. Para ello, convertiremos los conjuntos $\mathbf{A_k}$ en trincas $\mathbf{T_k}$. Es decir, una trinca $\mathbf{T_k}$, son tres naturales ordenados, donde se cumple (3). La equivalencia de (1) hacia (3) viene descrito en la bibliografía.

$$T_k \sim (T_k[0], T_k[1], T_k[2]) \tag{3}$$

Una vez generadas las trincas como en (3), nos interesa formatearlas de manera encadenada $\mathbf{E_k}$ según (4), sin que pierdan la variabilidad original, y sin incluir por ello caso alguno.

$$\forall k\, E_k[2] = E_{k+1}[0] \tag{4}$$

Para conseguir la transformación sin añadir ni eliminar casos de (3) a (4), basta con ordenar los T_k de manera que, para cada k' que cumpla en $\mathbf{T_{k'}}$ la propiedad (4), diremos que $\mathbf{E_{k'}} = \mathbf{T_{k'}}$. Si, por el contrario, el componente 2 no iguala al componente 0 del sucesor, entonces se aplica la transformación (5).

$$\begin{gathered} \forall k\, T_k[2] \neq T_{k+1}[0] \rightarrow (T_k, T_{k+1}) \sim (T_k, E_1, E_2, E_3, T_{k+1}) \\ E_2[1] \in B \end{gathered} \tag{5}$$

Gracias a este enfoque, las trincas encadenadas se ajustan a un formato donde cada nodo se clasifica según si es par o impar. Salvo en los extremos, los pares representan un puente entre dos trincas, mientras que los impares el valle entre los dos puentes. Visto así, cuando dos nodos aparezcan en dos trincas, interesa denotarlo como si fuera un puente largo, cada puente largo será denominado con un número identificativo mayor que 1. De esa manera, tendremos la siguiente transformación necesaria: la que transforma las trincas ordenadas en una trinca (**P**, **S**, **D**), donde **P** son los pares **N x N** que contienen la correspondencia entre una *posición* y su *descriptor* según (6), **S** es el número de trincas encadenadas y **D** es el número de descriptores necesarios.

$$\begin{gathered} \forall k \geqslant 0\, \forall i \in \{0,1,2\}\, E_k[i] \sim (2 \times k + i, \mathit{descriptor}(E_k[i])) \\ \mathit{descriptor}(X) = \left\{ \begin{array}{c} 1 \leftarrow X \in B \\ 0 \leftarrow X \notin B \\ N \leftarrow \exists k\, \exists l\, k \neq l\, X \in E_k \wedge X \in E_l \end{array} \right\} \\ \forall X\, \forall Y\, \mathit{descriptor}(X) = \mathit{descriptor}(Y) > 1 \rightarrow X = Y \end{gathered} \tag{6}$$

Desde el formato propuesto por la trinca (**P**, **S**, **D**), ya estamos en disposición de generar una lista de quíntuplas que nos permita calcular el número de casos. Una quíntupla se compone de (**l**, **r**, **t**, **n**, **h**), que viene del inglés: *length, rattle, tail, neck, head*. Intenta interpretar una lista de elementos incluyéndole algo más que la cabeza y la cola. El primer valor nos dice cuántas trincas encadenadas contiene, mientras que los siguientes son los dos valores de las posiciones más bajas para **r** y **t** respectivamente, mientras que los valores de las posiciones más altas son **h** y **n** respectivamente. Estos valores sólo pueden ser los definidos por la función descriptor en (6). En el listado 1 observamos un algoritmo que transforma a partir de los pares, el último par y el último descriptor usado en la lista de quíntuplas equivalente. Como podemos observar, el listado 1 es O(**S**), sin olvidar que **S** es proporcional al número de $\mathbf{A_k}$ de (1).

El formato de listado de quíntuplas.

Ahora que ya tenemos la estructura necesaria para contabilizar el número de casos, interesa saber cuál es la función a través de la cual girarán todos nuestros resultados. Esta función está presentada en el listado 2, partiendo de que le damos valor a la longitud y que, de no conocer el valor booleano de **r**, **t**, **n** y **h**, lo dejaremos con el valor **nulo**, entonces la función nos devuelve a través de una recurrencia simple, el número de casos. Esta función se puede optimizar por dos vías, por un lado se le puede quitar la recurrencia, para volverla iterativa y, por el otro lado, si bien esta función hace una llamada a la función de Fibonacci, es bien conocido que esta sucesión es implementable mediante un exponencial a razón del número aúreo. Sin embargo, esas pesquisas se salen de la

intencionalidad de este documento.

El listado 3 nos muestra las dos clases que vamos a necesitar para desguazar los descriptores sin que ello nos lleve a una explosión combinatoria. La clase *Piel* usa la clase *Contexto*, *Piel* se inicializa con los valores de la quíntupla definida por la función descriptor en (6). Ahora bien, al ser una clase, se reserva el derecho de atribuir a los cuatro valores un valor booleano específico: 0 o 1 a través de una función __call__. El asignar las 16 combinaciones booleanas sobre los valores **r**, **t**, **n** y **h**, mediante un entero del 0 al 15 supone, a su misma vez, atribuirle a los descriptores un valor booleano, y es así como queda reflejado a través de la clase *Contexto*.

Por tanto, el cometido de la clase *Contexto* es almacenar las asignaciones que deben adoptar los descriptores siempre y cuando éstas contabilicen al menos un caso coherente. El uso de la clase Contexto es, para colocar en el descriptor especial None el número de casos contemplados por ese grupo de asignaciones, por lo que la combinación de asignaciones compatibles será el álgebra en el que se centren los distintos contextos. Nuestra álgebra de sumas y productos están definidos en la clase *Contexto* mediante los métodos __add__ y __mul__.

Ciertamente, la suma equivale a añadir más casos, mientras que el producto equivale a restringir los dos contextos multiplicando el número de casos entre sí. Para que el producto se pueda aplicar los extremos de las quíntuplas deben haber quedado previamente asignados con un valor pertinente, y esa era la razón por la cual sólo se podía usar *Contexto* desde *Piel,* para que Piel aislara con asignaciones cada subsecuencia de trincas.

Esto nos lleva al listado 4, donde está desarrollada la función que manejará las 16 combinaciones posibles, entendiendo que en cada momento el invariante consiste en reconocer que hemos leído una secuencia de trincas definido por un contexto donde los extremos **r**, **t**, **n** y **h** ya están predeterminados. Eso quiere decir que la función del listado 4 vincula bajo todas las combinaciones compatibles los posibles casos que nos permiten estudiar la siguiente quíntupla. Si bien el número de sumandos asciende a 64, ese valor se mantiene constante e independiente del tamaño de la entrada original. Por lo que al final el listado 4 nos devuelve el número de casos dentro de una O(**S**).

Conclusiones

Como se ha podido comprobar, existe una manera de trabajar con conjuntos explícitos reconociendo unos cardinales que trabajan como si fueran números grandes, sin sucumbir a la explosión combinatoria. Esto da pie a pensar que deban existir una enorme gama de problemas que aún ni nos imaginábamos que eran rápidos de tratar. De hecho, estas técnicas dan constancia del enorme problema de codificación de cierta clase de problemas antes que de resolución, pues al final bien podría resolverse linealmente si a través de los cálculos intermedios no necesitamos aumentar la complejidad. Así que, ¿hasta qué punto podría modificarse el algoritmo para que sea incluso aún más rápido? Se trata de una cuestión para la cual no hay ser humano con inventiva posible que pueda dar una respuesta concluyente.

Referencias

(las mías)

```python
#pares es extrictamente ascendente en la primera componente
def _pieles(pares, ultPar, ultAux):
    if not pares:
        return [(ultPar//2,None,None,None,None)]
    R=[]
    if not pares[0][0]==0:
        pares.insert(0,(0,None))
    while pares:
        r = pares[0][1]
        if len(pares)<2:
            l=(ultPar-pares[0][0])//2
            R.append((l,r,None,None,None))
            return R
        elif pares[1][0]%2==1:
            if pares[1][0]==pares[0][0]+1:
                t = pares[1][1]
                if len(pares)<3:
                    l=(ultPar-pares[0][0])//2
                    R.append((l,r,t,None,None))
                    return R
                elif pares[2][0]%2==1:
                    n = pares[2][1]
                    if len(pares)<4:
                        l=(pares[2][0]+1-pares[0][0])//2
                        R.append((l,r,t,n,None))
                        R.append(((ultPar-pares[2][0]-1)//2,
                                  None,None,None,None))
                        return R
                    elif pares[3][0]==pares[2][0]+1:
                        h = pares[3][1]
                    else:
                        ultAux += 1
                        h = ultAux
                        pares.insert(3,(pares[2][0]+1,h))
                    l = (pares[2][0]+1-pares[0][0])//2
                    pares.pop(0)
                    pares.pop(0)
                    pares.pop(0)
                else:
                    n = None
                    h = pares[2][1]
                    l = (pares[2][0]-pares[0][0])//2
                    pares.pop(0)
                    pares.pop(0)
            else:
            else:
                t = None
                n = pares[1][1]
                if len(pares)<3:
                    l=(pares[1][0]+1-pares[0][0])//2
                    if pares[1][0]+1<ultPar:
                        R.append((l,r,t,n,ultAux+1))
                        R.append(((ultPar-(pares[1][0]+1))//2,
                                  ultAux+1,None,None,None))
                    else:
                        R.append((l,r,t,n,None))
                    return R
                elif pares[2][0] == pares[1][0]+1:
                    h = pares[2][1]
                else:
                    ultAux += 1
                    h = ultAux
                    pares.insert(2, (pares[1][0]+1,h))
                l = (pares[1][0]+1-pares[0][0])//2
                pares.pop()
                pares.pop()
        else:
            t = None
            n = None
            h = pares[1][1]
            l = (pares[1][0]-pares[0][0])//2
            pares.pop()
        R.append((l,r,t,n,h))
    return R
```

Listado 1

```
def S(lon, cascabel=None,cola=None,cuello=None,cabeza=None):
    if lon==1: #cola==cuello (cascabel|cola|cabeza)=1
        if cola is None and not cuello is None:
            cola=cuello
        elif not cola is None and cuello is None:
            cuello=cola
        elif not cola==cuello:
            return 0
        if cola==1 and (cabeza==1 or cascabel==1):
            return 0
        elif cola==0 and cabeza==0 and cascabel==0:
            return 0
        elif cabeza==1 and cascabel==1:
            return 0
        if cabeza==1 or cuello==1 or cascabel==1:
            return 1
        return int(cabeza is None)+int(cuello is None)+int(cascabel is None)
    else:
        if cabeza==1:
            return S(lon-1,cabeza=cascabel, cuello=cola, cascabel=0) \
                   if not cuello==1 else 0
        elif cabeza==0:
            return S(lon-1,cabeza=None if cuello is None else 1-cuello,
                     cola=cola, cascabel=cascabel)
        elif not cuello is None:
            return S(lon,cascabel=cascabel,cola=cola,cabeza=cuello)
        elif cascabel is None and cola is None:
            return fib(lon+2)
        else:
            return S(lon,cabeza=cascabel, cuello=cola)
```

Listado 2

```python
class Contexto:
    def __init__(self, c={None:0}):
        self.cuerpo=c
    def __repr__(self):
        return "Contexto("+repr(self.cuerpo)+")"
    def __bool__(self):
        return self.cuerpo[None]>0
    def __getitem__(self, key):
        return self.cuerpo[key]
    def __setitem__(self, key, item):
        self.cuerpo[key]=item
    def keys(self):
        return self.cuerpo.keys()
    def __add__(self, other):
        if not self:
            return other
        if not other:
            return self
        R=Contexto({None:(self[None]+other[None])})
        for x in self.keys():
            if x is None:
                continue
            if not x in other.keys():
                pass
            elif self[x]==other[x]:
                R[x]=self[x]
        return R
    def __mul__(self, other):
        if not self or not other:
            return Contexto()
        R=Contexto({None:(self[None]*other[None])})
        for x in self.keys():
            if x is None:
                continue
            if not x in other.keys():
                R[x]=self[x]
            elif self[x]==other[x]:
                R[x]=self[x]
            else:
                return Contexto({None:0})
        for x in other.keys():
            if x is None:
                continue
            if not x in self.keys():
                R[x]=other[x]
        return R
```

```python
class Piel:
    '0 e 1 son valores, mientras que >1 son descriptores'
    def __init__(self, l, r=None,t=None,n=None,h=None):
        self.r=r #cascabel
        self.t=t #cola
        self.n=n #cuello
        self.h=h #cabeza
        self.l=l #longitud

    def __call__(self, X):
        'Número del 0 al 15'
        R=int(X>=8)
        X%=8
        T=int(X>=4)
        X%=4
        N=int(X>=2)
        H=X%2
        contexto={}
        for a,A in (self.r,R),(self.t,T),(self.n,N),(self.h,H):
            if not a is None:
                if a>1:
                    contexto[a]=A
                elif not a==A:
                    return Contexto()
        contexto[None]=S(self.l,R,T,N,H)
        return Contexto(contexto)
```

Listado 3

```python
def peleteria(*pieles):
    W=[pieles[0](k) for k in range(16)]
    for Sk in pieles[1:]:
        V=[Contexto({None:0}) for k in range(16)]
        for i in range(4):
            for x in range(16):
                V[4*i+x%4]+=W[4*i+int(x>=8)]*Sk(x)+\
                             W[4*i+int(x>=8)+2]*Sk(x)
        for k in range(16):
            W[k]=V[k]
    Sum=Contexto()
    for X in W:
        Sum+=X
    return Sum[None]
```

Listado 4